LiveCore: Increasing Liveness in a Low-Level Dataflow Programming Environment

David Alexander
Toybox
London, UK
davidleonalex@gmail.com

Jack Armitage
Centre for Digital Music
Queen Mary University of London
j.d.k.armitage@qmul.ac.uk

ABSTRACT

Liveness is an important factor in live coding but frequently liveness focuses on high-level, textual environments. While these environments offer manifold abstraction capabilities, users of low-level dataflow programming environments could also benefit from increased liveness. In this work we introduce LiveCore: a macro library for the low-level dataflow environment Reaktor Core enabling live coding. LiveCore manages program state at audio rates and provides a suite of modules for musical pattern generation, sequencing and synthesis. LiveCore increases liveness in Reaktor Core from an editable dataflow program, to one with continuous audio suitable for musical performance. We reflect on the design process to discuss the qualitative differences of liveness in low-level dataflow programming, compared with other forms of live coding. We suggest that live coding in a low-level dataflow environment provides a uniquely immediate experience for the performer.

CCS CONCEPTS

• **Applied computing** → **Sound and music computing**; *Performing arts*; • **Human-centered computing** → *User interface programming*.

KEYWORDS

live coding, low-level languages, dataflow programming, visual programming

ACM Reference Format:
David Alexander and Jack Armitage. 2019. LiveCore: Increasing Liveness in a Low-Level Dataflow Programming Environment. In *Audio Mostly 2019, September 18–20, 2019, Nottingham, UK*. ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/1122445.1122456

Audio Mostly '19, September 18–20, 2019, Nottingham, UK

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Tanimoto defines the essential characteristic of liveness being simply the ability to modify a running program [8], but goes on to describe multiple levels which ascend in their sophistication. Liveness is an essential aspect of live coding, a term used both by non-artistic programming research communities such as the LIVE Workshop[1], and also by artist programmers [6] engaged in live performance such as the International Conference on Live Interfaces (ICLI)[2].

Live coding is quite often paired with a high-level, text-based programming environment and language, with prominent examples from computer music being SuperCollider [5] and TidalCycles [6]. When referring to programming languages, the level refers to the number of layers of abstractions between the computational substrate and the programmer. Exact levels are rarely quantified, and whether a language is considered low or high-level changes as technological trends evolve. In this work we define a low-level programming language as one that provides few or no abstraction layers on top of the instruction set of the computer. Dataflow programming languages such as Pure Data [7] provide a diagrammatic paradigm for program representation using the metaphor of boxes and wired connections. Unlike in text-based languages this can make some programs visually understandable at a glance, however dataflow languages typically introduce other constraints [3].

In this work we are specifically interested in exploring the meeting point of liveness, low-level languages, and dataflow programming. In doing so we aspire to a particular form of "embodiment of code" with similarities between the "code embodied by the human" and the "code embodied by the machine" [1]. To use an analogy, we seek a musical programming experience with the same immediacy and directness felt by a guitarist as their fingers slide across the strings and their instrument vibrates through their body. The next section of this paper describes *Reaktor Core*, the target programming environment for live coding in this project. The following

[1] https://2019.splashcon.org/home/live-2019
[2] http://www.liveinterfaces.org

section then details the design of LiveCore. Finally, we reflect on our experience of designing and using LiveCore.

2 BACKGROUND

The Reaktor Core dataflow audio language

Reaktor Core, hereafter referred to as simply *Core* is a compiled low-level dataflow language designed for efficient real-time audio digital signal processing (DSP) development that is integrated into *Reaktor*, a commercial closed source audio development tool from Native Instruments based on a visual dataflow paradigm [3]. An additional development layer available within Reaktor is *Primary*; a high-level event-driven collection of pre-compiled audio processing and control modules. However this layer is unsuitable for live coding since the audio engine stops passing audio signals when adding new Primary modules.

In the Core environment, compilation time is practically instant for simple projects and recompilation does not interrupt the audio engine, meaning real-time audio graph changes are possible. The Core programming language consists of a simple set of basic low-level operations roughly equating to central processing unit (CPU) operands such as add, multiply, and read and write to memory. The Core programming environment is self-contained with no calls to external libraries. Units of compiled Core code, known as *cells*, interface with audio and event input and output ports. Core's dataflow programming paradigm means the entire audio graph is always visualised as it runs, and it is simple to debug in real-time using a built-in wire debugging tool.

Unique aspects of coding in Reaktor Core

Compared with other dataflow style audio programming environments like Max or PureData, Core is low-level with no pre-compiled modules. All macros are written using the same limited set of operands, and so the distinction between sound generation and pattern manipulation is blurred. There is no abstraction between musical patterning, the audio graph, and the individual audio DSP components generating and manipulating the signals (filters, sample playback, synthesis etc). For example, sequencers can run at any speed up to the current audio clock frequency, so slow musical patterns become synthesiser waveforms when pitched up to audio frequencies.

Challenges for live coding in Reaktor Core

In Core, the program state is lost with each recompile. There is no facility for iteration as the graph is purely procedural and linear in structure, executed as a single stream of audio rate events. The interface is not optimised for manipulation of data in real-time. For example, changing values of constants requires multiple mouse clicks (enable edit mode, then enter the value) which can be tedious compared to using a text editor. Although there is a free player version of Reaktor it is only possible to edit in Core using the commercial version of the software. In addition there is no way to export or automatically port the code to other platforms due to the proprietary format.

[3] https://www.native-instruments.com/en/products/komplete/synths/reaktor-6/

3 DESIGN

LiveCore: a live coding library in Reaktor Core

LiveCore[4] is a library of modules for live coding in Core and consists of macros or *Blocks* containing code with sets of input and output ports. Each macro contains as few operations as possible thus keeping the code efficient and compact. Macros that are connected and therefor should execute are compiled into a few dozen bytes each. Disconnected macros are ignored by the compiler. The result is a single block of executable code that can sit inside the CPU's program cache without any calls to external functions.

The fundamental building block of the library is a *Phase Driver*: a ramp waveform generator that represents a musical period, which could be for example two measures long. The Phase Driver is based on a simple phase accumulator that increments every audio clock and wraps back to 0 when it reaches 1. Additional blocks can subdivide this cycle into smaller periods, which can be quantised and used to look up tables of values for parameter modulation or note data. The output of the ramp generator can be modified using a waveshaper to achieve different grooves or more complex rhythmic patterns. It can also function as an audio-rate oscillator by using a further waveshaper to create different wave-forms or by using it to read samples from a wavetable using the Sample Reader described later in this section. If wrapping is disabled, ramps can be triggered in oneshot mode to drive sample playback or be used as envelopes if followed by a function generator or another waveshaper.

Maintaining state in Core

The program state is saved every audio clock tick. Each module of code containing state is given a unique identifier that represents an offset into a memory stack. This stack is backed up to a table in Reaktor's Primary layer, outside of the Core environment. When the LiveCore program has finished recompiling, the saved state is immediately restored, resulting in a continuous audio experience.

Modules

Phase Driver. Ramp generator that can run in 'one-shot' or 'looping' mode.

[4] https://github.com/freeeco/livecore

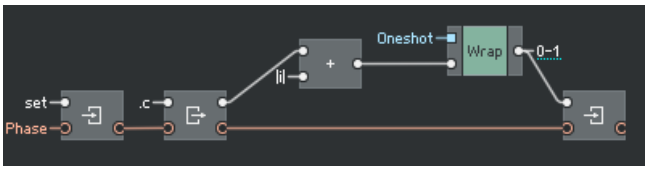

Figure 1: LiveCore Phase Driver module.

Sequencer. Quantises the output of the Phase Driver and uses the result to select pattern data from it's inputs.

Phase Splitter. Divides an input phase into sub-phases.

Gate. Creates a gate from a Phase Driver using the function "out = x < gate length".

Mixer. A simple module that sums together all the inputs.

Slew Limiter. Creates an up-ramp until the level at the input is reached and then a down-ramp. Triggered from a Gate it can be used as an envelope, and it can also be used as a signal smoother or filter.

Waveshaper. Creates different oscillator shapes from the output of the Phase Driver. It can be placed before sequencers to create tempo modulations or accelerations, simulate jitter and swing or otherwise alter pattern timing. It can also be placed between the Phase Driver and the Sample Reader to modify for example playback speed or direction.

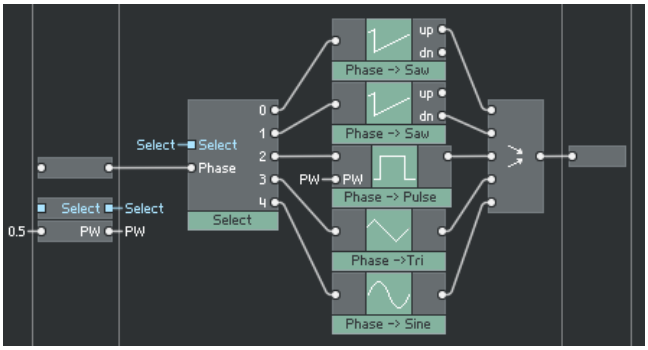

Figure 2: LiveCore Waveshaper module.

Sample Reader. Reads samples from a table and has inputs for 'Table Reference' and 'Position'. Attaching a Phase Driver to the position input will play through the sample. Short samples can be used as wavetables for more complex oscillator shapes than the Waveshaper.

Live coding workflow

Connecting several Phase Drivers together with a Phase Splitter quickly leads to musical patterns. These sub-ramps can then drive sequencer modules that read sequence data to trigger samples and other Phase Drivers. Synthesisers can be constructed using Phase Drivers and Waveshapers, that in turn can be combined to create oscillators and envelopes. The musical structure is modified in real-time by connecting new wires, changing the orders of the modules and modifying the data used by the sequences.

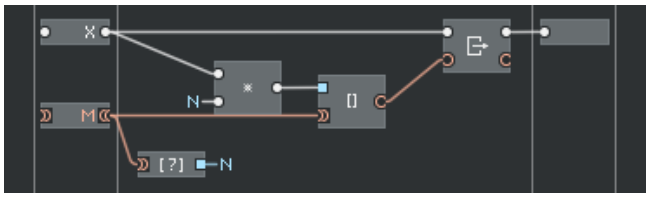

Figure 3: LiveCore Sample Reader module.

To avoid repetitive mouse clicks to enter program constants, additional control inputs can be integrated into the workflow. Combining LiveCore with the high-level pattern selection blocks or data input blocks, including a recordable X-Y Modulation Pad for gesture recording, allows for a more playable environment.

Performance and limitations

Saving and restoring program state adds overhead to execution. Although the compiled code runs efficiently the overhead limits the amount of modules to around 60 or 70 before the CPU overhead becomes considerable. A more efficient solution would be to have the program stack left un-initialised after compilation. This would require altering the way the Core compiler works and isn't currently possible due to it being closed source.

To reduce overhead in larger patches, the program state saving rate can be reduced from every audio clock to every N clocks. This has the disadvantage that small timing errors could potentially be introduced when the state is restored. These errors in practice are negligible if N is considerably small, with state saving occurring every four or eight audio clocks being a good compromise. Hosting multiple instances in a digital audio workstation (DAW) to mix between sections of the music is another simple way to spread CPU load across multiple cores. This approach has the additional compositional advantage of compartmentalising musical sections, allowing the loading of one section of music while another is playing back.

4 DISCUSSION

LiveCore demonstrates the main functionality that this project set out to achieve, in combining live coding, low-level audio programming, and dataflow programming. It provides a simple yet effective approach for maintaining state in a low-level environment as the audio graph is recompiled. While there are certain performance limits, these do not prevent an

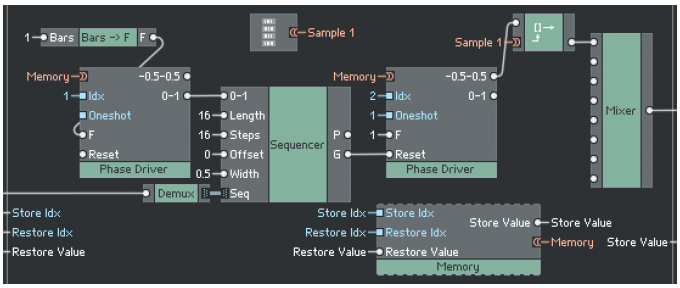

Figure 4: An example LiveCore program that shows a basic 16 step sequence driving a Sample Player into a Mixer.

author from creating a musical LiveCore patch. Here each author provides a first-person reflection based on making music together using LiveCore.

Second author; experienced TidalCycles live coder. Being used to high-level textual live coding, it was hard for me to imagine where to start with LiveCore. I first proposed something familiar; a one bar percussive loop using the step sequencer and sampler. As soon as we transformed sequences with the *Phase Splitter*, it immediately felt like live coding; program changes as questions with surprisingly musical answers. The surprises were qualitatively different however when feeding mixer audio outputs back into the sequencers, which is not possible in Tidal. Incredibly rich timbre and blurring of pattern and sound were instantly on hand.

First author; LiveCore designer. Even though I understood the mechanics of each module, as we introduced feedback there was a large amount of unpredictability and randomness. It felt like we were working directly with the computer's circuitry. Having the program completely exposed and self-contained we could try to envision what kind of unsolvable equation we had built while listening back to the textured musical patterns.

During the practice-based evaluation session, a bug was found in one of the macros which affected the storing program state. Recompilation became audible, but it turned out this could be musical too. This led to two ideas: performative recompilation and memory modulation. In performative recompilation, state changes are achieved by addding objects to the graph at specific times, which do not even need to be connected to the graph. Memory being situated in Core itself means that it would be highly unlikely this would cause crashes, opening up the possibility of modulating the program memory directly.

One of the main limitations of LiveCore is being tied to Reaktor Core. However exploring within this constraint led to us discovering some features that could be re-implemented in a more accessible context. As well as the macros themselves and the phase-based musical pattern processing, the performative recompilation and memory modulation ideas above could also be explored in a future reimplementation of LiveCore. We would not have necessarily landed on these ideas if we were using another environment.

By choosing to develop a low-level tool, the artist is given the opportunity to understand more or less the full extent of their program. Like a simple instrument carved from wood, it lends itself an unusual durability ("long-lived, stable, and if degrading, degrading gracefully") [2] due to its simplicity, as the artist can easily rework the simple routines into new environments as needed. Despite this, we believe there are subtle qualitative factors of immediacy at play even with a simple tool such as LiveCore.

Although LiveCore provides a stripped back programming experience compared to higher-level languages, it is nonetheless rife with constraints and possesses its own creative biases that influence the artist. The LiveCore library macros enable one to program structures in real-time, yet even these basic macros places their own limitations on what can be done creatively when performing. Since creating alternative macros on-the-fly would be too slow, the artist is limited by the ones that are available. In a way, the macros somewhat reflect implementations found in other higher-level composition environments that seek to reify traditional musical ideas, such as numerical structures based on factors of two [4]. Returning to Baalman's questions of how to "allow for a different way of embodying the code" [1], tools such as LiveCore highlight the usefulness of finding a meeting point between technical simplicity and aesthetic constraint.

ACKNOWLEDGMENTS

REFERENCES

[1] Marije Baalman. 2015. Embodiment of Code. In *Proceedings of the First International Conference on Live Coding*. ICSRiM, University of Leeds, 35–40. https://doi.org/10.5281/zenodo.18748

[2] Antranig Basman. 2016. Building Software Is Not a Craft. *Proceedings of the Psychology of Programming Interest Group* (2016).

[3] Daniel D. Hils. 1992. Visual Languages and Computing Survey: Data Flow Visual Programming Languages. *Journal of Visual Languages & Computing* 3, 1 (1992), 69–101.

[4] Thor Magnusson. 2009. Of Epistemic Tools: Musical Instruments as Cognitive Extensions. *Organised Sound* 14, 02 (2009), 168–176.

[5] James McCartney. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal* 26, 4 (2002), 61–68.

[6] Alex McLean. 2011. *Artist-Programmers and Programming Languages for the Arts.* Ph.D. Dissertation. Goldsmiths University of London, London, United Kingdom.

[7] Miller Puckette et al. 1996. Pure Data: Another Integrated Computer Music Environment. *Proceedings of the Second Intercollege Computer Music Concerts* (1996), 37–41.

[8] Steven L. Tanimoto. 2013. A Perspective on the Evolution of Live Programming. In *International Workshop on Live Programming*.